The Tree Representation of Feasible Solutions for the TSP with Pickup and Delivery and LIFO Loading

Dejian Tu and **Songshan Guo**
Department of Computer Science, School of Information Science and Technology,
Sun Yat-Sen University, Guangzhou, Guangdong, PR China (510275)
tudejian@gmail.com, issgssh@mail.sysu.edu.cn

Hu Qin and **Wee-Chong Oon** and **Andrew Lim**
Department of Management Sciences, City University of Hong Kong,
Tat Chee Ave, Kowloon Tong, Hong Kong
{tigerqin, weecoon, lim.andrew}@cityu.edu.hk

Abstract

The feasible solutions of the traveling salesman problem with pickup and delivery (TSPPD) are represented by vertex lists in existing literature. However, when the TSPPD requires that the loading and unloading operations must be performed in a last-in-first-out (LIFO) manner, we show that its feasible solutions can be represented by trees. Consequently, we develop a variable neighbourhood search (VNS) heuristic for the TSPPD with last-in-first-out loading (TSPPDL) involving several search operators based on the tree data structure. Experiments show that our VNS heuristic is superior to the current best heuristics for TSPPDL in terms of both solution quality and computing time.

Introduction

In the traveling salesman problem with pickup and delivery (TSPPD), there is a set of n demands, denoted by $R = \{1, \ldots, n\}$, each of which is composed of a pickup vertex and a delivery vertex. Let $P = \{1^+, \ldots, n^+\}$ be the set of pickup vertices and $D = \{1^-, \ldots, n^-\}$ be the set of delivery vertices. Vertex 0^+ and 0^- represent the exit from and the entrance to the depot, respectively. The TSPPD is defined on a complete and undirected graph $G = (V, E, d)$, where $V = P \cup D \cup \{0^+, 0^-\}$ is the vertex set; $E = \{(x, y) : x, y \in V, x \neq y\}$ is the edge set; and $d(x, y)$ denotes the non-negative distance between vertex x and y. The objective of the TSPPD is to find a shortest Hamiltonian tour on G, starting from vertex 0^+ and ending at vertex 0^-, for a vehicle with unlimited capacity, subject to the precedence constraint that each pickup vertex is visited before its associated delivery vertex. In most existing literature on the TSPPD, feasible solutions are represented by lists (or sequences) of vertices.

This study addresses a variant of the TSPPD in which loading and unloading operations must be performed in a last-in-first-out (LIFO) manner; this problem is referred to as the TSPPD with LIFO loading (TSPPDL). The TSPPDL has been considered a more complex problem than the TSPPD because both the precedence and LIFO constraints must be checked to ensure solution feasibility. In this paper, we show that there we can represent feasible solutions of TSPPDL using a tree, which greatly simplifies this problem. As a result, we were able to build upon the Variable Neighbourhood Search (VNS) heuristic proposed by (Carrabs, Cordeau, and Laporte 2007), which is the best existing approach for TSPPDL at the time of this writing; we reproduce four of the original operators using the tree representation, and also introduce three new operators. Our new heuristic outperforms the original in terms of both solution quality and computation time.

Existing Research

The TSPPDL was first mentioned by (Ladany and Mehrez 1984). However, they neither formulate it mathematically nor propose solution procedures except for enumeration.

(Carrabs, Cerulli, and Cordeau 2007) introduced a branch-and-bound algorithm that applies an additive lower bound technique proposed by (Fischetti and Toth 1989); this technique is able to solve all instances with 15 requests and several instances with 21 requests. Currently, the best exact algorithm for the TSPPDL is the branch-and-cut algorithm described in (Cordeau et al. 2010), which is based on the fundamental component from the commercial integer programming solver *ILOG CPLEX*; it is able to handle most instances with up to 17 requests in less than 10 minutes, and several instances with 25 requests within 1 hour.

For the generation of near-optimal solutions for the large instances widely encountered in practice, the best approaches so far have made use of efficient heuristics. (Cassani and Righini 2004) developed a greedy heuristic and a variable neighbourhood descent (VND) algorithm which combines four types of exchange operators. (Carrabs, Cordeau, and Laporte 2007) built on this work by devising a variable neighbourhood search (VNS) heuristic which included the four exchange operators along with four new operators. They compared the VND and VNS heuristics by solving instances with up to 375 requests; the computational results showed that at the expense of more computing time, the VNS heuristic produces significantly better solutions than the VND heuristic. Both implementations represented feasible solutions by vertex lists.

The Tree Representation of Feasible Tours

In this section, we describe the tree representation of feasible tours for the TSPPDL, which is the primary contribution of this study. In particular, a feasible tour for the TSPPDL can be represented as an ordered tree (i.e., there is an order for the children of each tree node) with $|R|+1$ nodes, where the root node is labeled 0 and the remaining nodes are labeled some permutation of $\{1, ..., |R|\}$; we call a tree of this type a *TSPPDL tree*. By representing solutions using this tree, the feasibility of the solution is automatically guaranteed.

Figure 1(a) shows an example of a TSPDDL tree. The dashed arrows in Figure 1(b) pictorially shows how this ordered tree can be converted into a tour that automatically respects the precedence and LIFO constraints of the TSPPDL; this is similar to a preorder traversal of the tree, where the pickup vertex is instantiated when its node is first encountered, and the delivery vertex is instantiated when the node is last encountered. The conversion procedure is provided in Algorithm 1, which runs in $O(n)$ time.

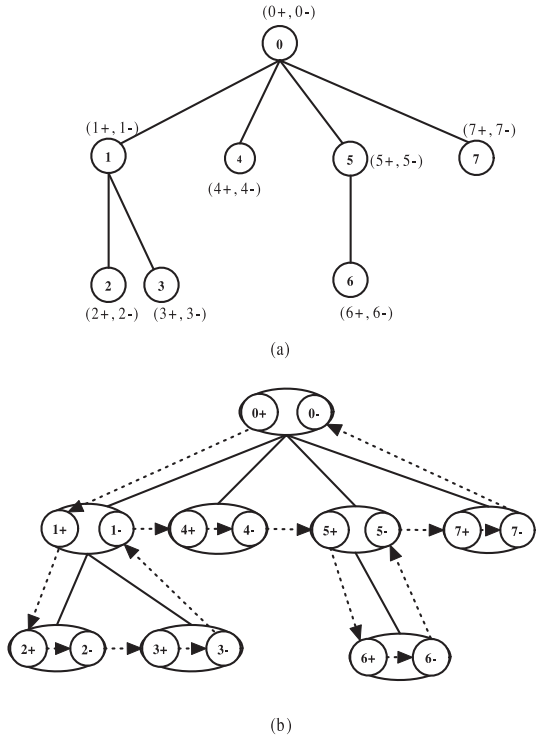

Figure 1: The tree representation of a feasible TSPPDL tour

We now define some terminology and notation. In this study we distinguish between the terms *node* and *vertex*: we specify that *node* refers to a TSPPDL tree node corresponding to a request in R, and *vertex* refers to the pickup or delivery vertex in V. We also define a *tour* as the vertex list representing a solution of the TSPPDL. Furthermore, we use the notation T_{S_x}, $x \in R$ to denote the subtree in T rooted at the node corresponding to request x. Finally, we will refer to TSPPDL trees simply as *trees*.

To show the correctness of our tree representation, we make use of the following property that is derived directly from the definition of TSPPDL:

Algorithm 1 Converting a TSPPDL tree into a feasible tour

```
1: INPUT: An ordered tree T;
2: Initialize the current node x ← node 0;
3: Initialize the current tour S ← ∅;
4: Execute recursive procedure DFS(T, S, x), defined as:
5: DFS(T, S, x)
6: {
7: Append x+ to the tail of S;
8: while node x has unvisited children do
9:    y ← the leftmost unvisited child of node x;
10:   Invoke DFS(T, S, y);
11: end while
12: Append x− to the tail of S;
13: }
14: Return S;
```

Property 1 *Let S be a tour and $pos(x)$ be the position of vertex x in S. The tour S is feasible to the TSPPDL if and only if any two requests (x^+, x^-) and (y^+, y^-) in S, $x \neq y$, satisfy one of the following four conditions: (1) $pos(x^+) < pos(y^+) < pos(y^-) < pos(x^-)$; (2) $pos(y^+) < pos(x^+) < pos(x^-) < pos(y^-)$; (3) $pos(x^+) < pos(x^-) < pos(y^+) < pos(y^-)$; (4) $pos(y^+) < pos(y^-) < pos(x^+) < pos(x^-)$.*

Theorem 1 *Let T be a TSPPDL tree. The tour generated from T using Algorithm 1 is a feasible solution to the TSPPDL.*

Proof. Any node in the tree can be seen as the root of a subtree. Consider any two subtrees T_{S_x} and T_{S_y} in the tree, $x \neq y$. (1) If $T_{S_y} \subset T_{S_x}$, then the generated tour must have $pos(x^+) < pos(y^+) < pos(y^-) < pos(x^-)$, which satisfies condition 1 of Property 1; (2) If $T_{S_x} \subset T_{S_y}$, then the generated tour must have $pos(y^+) < pos(x^+) < pos(x^-) < pos(y^-)$, which satisfies condition 2 in Property 1; (3) If $T_{S_y} \cap T_{S_x} = \emptyset$, then all requests in subtree T_{S_x} must be fulfilled either before or after requests in T_{S_y}, and accordingly the generated tour must have either $pos(x^+) < pos(x^-) < pos(y^+) < pos(y^-)$ or $pos(y^+) < pos(y^-) < pos(x^+) < pos(x^-)$, which satisfies either condition 3 or 4 in Property 1. Since the relative positions of any two subtrees T_{S_x} and T_{S_y} in the tree must belong to one of the above three possibilities, and each possibility must satisfy one of the four conditions described in Property 1, the theorem holds. □

To convert a feasible tour into a tree, a reverse procedure is presented in Algorithm 2, which is the inverse of Algorithm 1 and also runs in $O(n)$ time.

Compared to the list representation, the advantages brought about by the tree representation are threefold. Firstly, when any tree-based search heuristic is applied to the TSPPDL, the feasibility of the solution is automatically guaranteed, which makes the development and implementation of tree-based search heuristics much simpler and more direct. Secondly, we can derive more search operators based

Algorithm 2 Converting a feasible tour into a TSPPDL tree

```
INPUT: A feasible tour S;
Let node(v) be the node associated with vertex v ∈ S;
Initialize the current vertex v_current ← 0+;
Initialize stack Q ← ∅;
Initialize the current tree T ← ∅;
while v_current is not 0− do
    if v_current is 0+ then
        Push v_current into Q;
    else if v_current ∈ P then
        Insert node(v_current) into T as the rightmost child
        of node(top(Q));
        Push v_current into Q;
    else
        Pop the top element of Q;
    end if
    v_current ← the successor of v_current;
end while
Return T;
```

on the tree representation compared to the list representation, which enables search heuristics to explore the feasible regions of the TSPPDL more thoroughly. Thirdly, the TSPPDL only deals with the one-vehicle case without considering factors such as the capacity of the vehicle nor the time window for pickup and delivery. Extensions of the TSPPDL to handle variations such as limited capacity or multiple vehicles can be naturally handled using the tree representation; the same cannot be said for the vertex list representation, which already requires complex implementations for the basic problem.

To the best of our knowledge, (Carrabs, Cordeau, and Laporte 2007) is the current best search heuristic for the TSPPDL. In that paper, ensuring the feasibility of solutions generated by operations caused much added complexity and computational effort because the vertex list representation offers no inherent assurance of feasibility. In comparison, we reproduce the four basic operators they employed using the tree representation in a comparatively simple and natural manner. Additionally, we introduce three new operators that are derived naturally from the tree representation and would have been difficult to reproduce using the vertex list representation. We will refer to the original VNS heuristic as VNS-List, while our heuristic will be called VNS-Tree.

Variable Neighbourhood Search Algorithm

Variable neighbourhood search (VNS) was introduced by (Mladenović and Hansen 1997) for solving complex combinatorial and global optimization problems. Many VNS-based heuristics have been successfully applied to a variety of problems, e.g., the TSP; the p-median problem; the multi-source Weber problem; and the minimum sum-of-squares clustering problem (Hansen and Mladenović 2001).

In this section, we define the seven search operators for our VNS heuristic, each transforming a given TSPPDL tree T into a new one T'. As each of the trees corresponds to a feasible tour, these operators can be viewed as functions changing a feasible tour S to another feasible tour S'. Hence, there is no need to consider the feasibility of solutions when applying these operators.

Basic Operators

There are four basic search operators in our VNS heuristic: *subtree-swap*; *node-swap*; *subtree-relocate*; and *node-relocate*. These operators have exactly the same effect as the *block-exchange*; *couple-exchange*; *relocate-block*; and *relocate-couple* basic operators described in (Carrabs, Cordeau, and Laporte 2007), respectively, differing only in the data structure used to represent the solutions. Similar operators have also been utilized to deal with other problems, although they were all applied to vertex lists rather than trees (Taillard et al. 1997; Li and Lim 2003; Bent and Van Hentenryck 2004).

It is apparent, however, that using the tree representation greatly simplifies the implementation of these operators compared to the vertex list representation. The *subtree-swap* operator selects two subtrees and swaps their positions; the *node-swap* operator swaps two nodes; the *subtree-relocate* operator moves a subtree of T to a different position.

The *node-relocate* operator is slightly more complex. Given a feasible tour S, the operation removes a request (x^+, x^-) from S such that a new tour S' is created. Then, the vertices x^+ and x^- are inserted separately into S' while preserving feasibility. The example in Figure 2 demonstrates how to perform this operation on a tree T. First, node x is removed from T and its children are linked to its parent node (Figure 2(b)) such that a new tree T' is created. Next, x is inserted into T', e.g., as the second child of node 0 as shown in Figure 2(c); this is equivalent to fixing the position of x^+ in S' such that it follows vertex 1^- in the corresponding tour. Suppose x now has β right siblings; the set of all possible positions of x^- in S' is equivalent to relocating the $0, \ldots, \beta$ right siblings of x as the children of x (Figure 2(c–f)).

In fact, we make use of the *multi-relocate* refinement introduced in the original VNS-List implementation in place of node-relocate, which stores all improving node relocations in a list, executes the best relocation, and then performs the other relocations in the list if they are still valid. This speeds up the node-relocate operator in practice, although the asymptotic running time remains $O(n^3)$.

New Operators

We introduce three new operators in VNS-Tree, all of which were aided in their design by the tree representation.

The *ATSP* operator views a set of subtrees of a node as an instance of the asymmetric TSP, and searches for the optimal order of the children of all nodes; we adapted the *Randomized Arbitrary Insertion* (RAI) algorithm proposed by (Brest and Žerovnik 2005) for this purpose. Our ATSP operation is given as follows. Let the ordered set of child subtrees for a node x in a tree T be $C(x) = \{c_1, c_2, ..., c_{|C(x)|}\}$, such that all c_k are subtrees. If the number of such subtrees is small ($|C(x)| \leq 7$), we simply enumerate all possible orderings and select the one that results in the shortest tour.

If $|C(x)| \geq 8$, we use an adapted RAI algorithm as follows. First, randomly remove from $C(x)$ a sequential subset

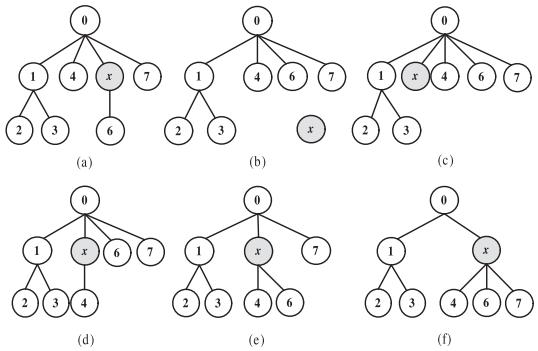

Figure 2: (a) The initial tree. (b) The new tree created after removing node x. (c)–(f) The new tree created by relocating node x.

$C'(x)$, i.e., $C'(x) = \{c_i, c_{i+1}, ..., c_j\}, 1 \leq i \leq j \leq |C(x)|$. Then, a component subtree in $C'(x)$ is randomly selected and reinserted into T as a child of x at the position resulting in the shortest tour; this is repeated for all component subtrees in $C'(x)$. The resultant solution is retained if it improves on the best solution found so far. This removal and reinsertion process is performed a total of n^2 times.

The *crossover* operator is inspired by the crossover operation in genetic algorithms. It is performed on two trees T_1 and T_2, which are called the primary and secondary tree respectively. First, we randomly remove a subtree T_S from T_1 and eliminate all edges in T_S to generate a graph consisting of a tree $T_1' = T_1 - T_S$ and a set of individual nodes V_S. Then, all the nodes in V_S are inserted into T_1' to construct a new tree T' abiding by the rule that each node in V_S must have the same parent in both T' and T_2.

Figure 3 provides an example of the crossover process. Given the two trees T_1 and T_2 in Figures 3(a) and (b), the randomly selected subtree T_S is selected and removed from T_1, resulting in the graph T_1' shown in Figure 3(c). Since the parent of node 1 in T_2 is node 3, the operation inserts node 1 into the graph as the child of node 3 (Figure 3(d)). The same criterion is used to insert nodes 2 and 3 into the graph, resulting in the final tree T' shown in Figure 3(f).

Finally, the *perturbation* operator randomly removes a subtree, and then greedily reinserts all the component nodes in the subtree back into the original solution. Given a tree T, we randomly select a node x and remove the subtree T_{S_x} from T, resulting in the partial solution $T' = T - T_{S_x}$. A node in T_{S_x} is randomly selected and reinserted into T' at the position resulting in the shortest tour; this is repeated for all nodes in T_{S_x}.

Variable Neighbourhood Search Heuristic

In our VNS-Tree heuristic, we maintain an array of solutions called *population* with *pop_size* elements. In the first iteration, we generate a random solution S_{best}. The *population* array is then initialized by performing the perturbation operation on S_{best} *pop_size* times.

For each of these solutions, we first perform the operations node-swap; subtree-swap; subtree-relocate; and multi-

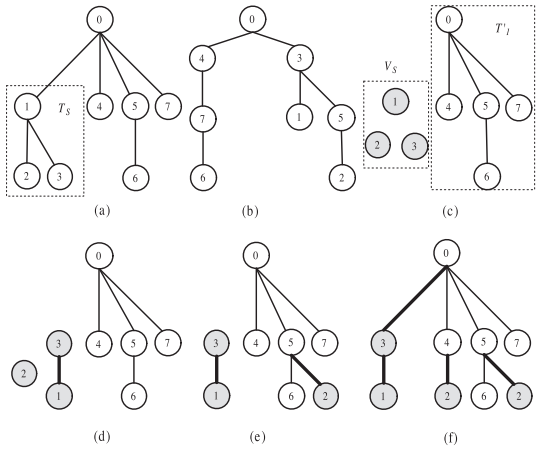

Figure 3: (a) T_1; (b) T_2; (c) The graph consisting of T_1' and a set of nodes V_S; (d)–(f) Insertion of V_S nodes into T_1'.

relocate in that order, restarting from the first operation whenever an improvement occurs. Next, we perform the ATSP operation. Then, for every pair of solutions we perform the crossover operation. Finally, we replace S_{best} with the best solution in the *population* array if it is superior; this marks an improving iteration. An iteration ends after we perform the perturbation operation on the elements in the *population* array; we repeat the process until *max_nonimproving* consecutive non-improving iterations occurs.

At this point, the *population* array is reinitialized by performing the perturbation operation on S_{best} *pop_size* times, and the search is performed again. This is done a total of *max_iter* times, whereupon the algorithm returns S_{best}.

Computational Experiments

We compared our VNS-Tree heuristic to the VNS-List heuristic over 96 TSPPDL instances in two categories (Types 1 and 2). The Type 1 data set was introduced in the original publication for the VNS-List heuristic; this data set was yielded from six TSP instances taken from TSPLIB (Reinelt 1991), which is a standard test suite for the TSP. For each of these TSP instances, subsets of vertices were selected with 25; 51; 75; 101; 251; 501; and 751 vertices, and the requests were randomly generated; we extended this test set by creating an additional test case with 1001 vertices for each instance. This generates 48 test instances.

The Type 2 data set was constructed using optimal TSP tours. To obtain this data set, we used the Type 1 vertex sets. We first find an optimal TSP tour S_{TSP} for the vertex set by running the Concorde TSP solver (Applegate et al. 2005). Next, we randomly construct a feasible TSPPDL tour, denoted by S_{TSPPDL}. Finally, we attach the label of each vertex in S_{TSPPDL} to its counterpart in S_{TSP}. In this way, we are able to generate 48 TSPPDL test instances with known optimal solutions.

The authors of VNS-List kindly supplied us with their source code, which was implemented in C. Our VNS-Tree algorithm was implemented in C++. Both algorithms were

Instance	Size	VNS-List Cost	VNS-List Time (s)	VNS-Tree Cost	VNS-Tree Time (s)	Gap(%)
fnl4461	501	71,604.0	335.78	68,876.6	337.74	3.81
	751	118,949.3	1,514.76	114,030.0	1,360.74	4.14
	1001	171,559.5	7,124.89	166,489.8	3,265.75	2.96
brd14051	501	52,959.0	339.31	50,369.9	391.21	4.89
	751	85,922.1	1,887.44	82,983.1	1,366.43	3.42
	1001	122,619.7	7,771.57	118,675.4	3,327.25	3.22
d18512	501	52,539.5	355.10	49,544.7	417.09	5.70
	751	84,205.5	1,823.17	80,734.1	1,319.35	4.12
	1001	122,925.8	7,051.67	118,675.4	3,328.77	3.46
d15112	501	954,071.7	325.23	926,331.2	395.52	2.91
	751	1,350,704.7	1,545.95	1,311,002.1	1,252.89	2.94
	1001	1,656,088.1	7,354.25	1,602,127.4	3,368.20	3.26
nrw1379	501	60,669.8	349.62	58,441.8	377.83	3.67
	751	105,495.7	1,668.71	101,737.7	1,407.51	3.56
	1001	161,465.0	6,511.14	155,471.9	2,930.06	3.71
pr1002	501	486,090.5	343.58	470,294.5	387.89	3.25
	751	816,215.3	1,560.94	788,885.5	1,363.88	3.35
	1001	1,159,345.6	7,636.01	1,122,976.0	3,099.81	3.14

Table 1: Performance Comparison Between VNS-List and VNS-Tree Based on Type 1 Data Set

run on a Linux server with 3 GB memory and Intel Xeon(R) 2.66 GHz processor. VNS-Tree has 3 parameters, which we fixed after some preliminary experimentation as follows: $max\_iter = 50$; $max\_nonimproving = 15$; and $pop\_size = 10$. Each instance was solved ten times, and the average tour costs and computing times were reported.

For the sake of brevity, we only provide the detailed results for the three largest instances with 501; 751; and 1001 vertices for each test set. For all the smaller instances with 25; 51; 75; 101; and 251 vertices, VNS-Tree produces equivalent or better solutions than VNS-List, and the gap in performance increases with problem size.

Table 1 gives the results for the Type 1 data set, where the last column gives the percentage gap between the solution values of VNS-Tree and VNS-List. The results show that VNS-Tree was able to find better solutions than VNS-List over all instances; the average percentage improvement is 2.56%. In fact, VNS-Tree found better final solutions than VNS-List for 40 out of the 48 instances, and produced the same solutions as VNS-List for the remainder; the eight instances where VNS-Tree did no better than VNS-List are small instances with either 25 or 51 vertices.

Table 2 presents the results for the Type 2 data set, where the second column gives the length of the optimal solution for each instance. The results show that the VNS-Tree algorithm was able to find optimal solutions for 40 out of the 48 instances, including all the instances with 251 vertices or fewer (not shown); furthermore, for the 8 instances where the optimal solution was not found (highlighted in bold), the difference from optimal is less than 0.5%. In contrast, VNS-List was only able to solve 20 out of the 48 instances; all of these instances had 101 or fewer vertices. The average improvement of VNS-Tree over VNS-List for this test set is 5.06%, and the results suggest that the gap will increase as the size of the problem instances increases.

For both test cases, our implementation of VNS-Tree runs faster than VNS-List as the size of the problem instances increases. However, this is likely to be only by a constant factor since the asymptotic running time for both algorithms is $O(n^3)$. In any case, our results suggest that VNS-Tree runs no slower than VNS-List while producing significantly better solutions.

Conclusions and Future Work

The main contribution of this study is the tree representation of feasible solutions for the TSPPDL. By using this data structure, the feasibility of the solution is automatically guaranteed, which aids greatly in both the design and implementation of search operators for the problem. We reproduced the basic search operators using the tree structure and designed several new operators for a VNS heuristic for the problem; experimental results revealed that our heuristic outperforms the previous best VNS-List heuristic.

This work opens up new directions of research for the TSPPDL. A major failing of the vertex representation is the need to ensure solution feasibility. This makes the implementation of certain techniques, e.g., genetic algorithms or simulated annealing, difficult or impossible. In contrast, the tree representation naturally lends itself to such approaches; in fact, the crossover operator used in VNS-Tree can be employed for the same purpose in an evolutionary algorithm. The tree data structure can also be considered for other problems involving pickup and delivery with the LIFO constraint, including the addition of capacity constraints; multi-

Instance	Size	Optimal Solution	VNS-List		VNS-Tree		Gap(%)
			Cost	Time (s)	Cost	Time (s)	
fnl4461	501	21,818.0	23,559.5	350.50	21,818.0	156.26	7.39
	751	30,079.0	33,373.2	3,482.52	30,079.0	412.80	9.87
	1001	**40,161.0**	48,290.8	14,853.43	**40,229.6**	937.92	16.69
brd14051	501	14,853.0	15,559.4	426.64	14,853.0	140.25	4.54
	751	**22,115.0**	25,186.8	3,464.85	**22,118.1**	383.29	12.18
	1001	**32,141.0**	35,593.5	7,660.13	**32,153.8**	938.86	9.66
d18512	501	14,853.0	15,917.5	408.17	14,853.0	141.15	6.69
	751	22,115.0	24,328.8	3,145.05	22,115.0	405.24	9.10
	1001	**32,141.0**	39,109.4	11,310.67	**32,293.9**	962.63	17.43
d15112	501	**271,463.0**	291,305.2	356.82	**271,519.3**	156.94	6.79
	751	334,413.0	375,759.5	2,738.65	334,413.0	422.89	11.00
	1001	**385,358.0**	493,592.7	8,213.00	**387,128.7**	1,069.67	21.57
nrw1379	501	18,598.0	21,543.3	452.43	18,598.0	158.62	13.67
	751	**29,228.0**	32,666.4	2,469.73	**29,280.0**	423.76	10.37
	1001	40,518.0	49,314.8	7,415.68	40,518.0	982.51	17.84
pr1002	501	134,803.0	153,012.7	315.94	134,803.0	140.34	11.90
	751	195,752.0	204,241.7	2,558.34	195,752.0	362.62	4.16
	1001	**258,231.0**	329,492.0	11,857.80	**258,270.9**	861.49	21.62

Table 2: Performance Comparison Between VNS-List and VNS-Tree Based on Type 2 Data Set

ple vehicles; and time windows.

References

Applegate, D.; Bixby, R.; Chvátal, V.; and Cook, W. 2005. Concorde tsp solver. http://www.tsp.gatech.edu/concorde/index.html.

Bent, R., and Van Hentenryck, P. 2004. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science* 38(4):515–530.

Brest, J., and Žerovnik, J. 2005. A heuristic for the asymmetric traveling salesman problem. In *Proceedings of The 6th Metaheuristics International Conference*, 145–150.

Carrabs, F.; Cerulli, R.; and Cordeau, J.-F. 2007. An additive branch-and-bound algorithm for the pickup and delivery traveling salesman problem with lifo or fifo loading. *INFOR: Information Systems and Operational Research* 45(4):223–238.

Carrabs, F.; Cordeau, J.-F.; and Laporte, G. 2007. Variable neighborhood search for the pickup and delivery traveling salesman problem with lifo loading. *INFORMS Journal on Computing* 19(4):618–632.

Cassani, L., and Righini, G. 2004. Heuristic algorithms for the tsp with rear-loading. In *35th Annual Cong. Italian Oper. Res. Soc. (AIRO XXXV)*.

Cordeau, J.-F.; Iori, M.; Laporte, G.; and Salazar González, J. J. 2010. A branch-and-cut algorithm for the pickup and delivery traveling salesman problem with lifo loading. *Networks* 55(1):46–59.

Fischetti, M., and Toth, P. 1989. An additive bounding procedure for combinatorial optimization problems. *Operations Research* 37(2):319–328.

Hansen, P., and Mladenović, N. 2001. Variable neighborhood search: Principles and applications. *European Journal of Operational Research* 130(3):449–467.

Ladany, S. P., and Mehrez, A. 1984. Optimal routing of a single vehicle with loading and unloading constraints. *Transportation Planning and Technology* 8(4):301–306.

Li, H., and Lim, A. 2003. Local search with annealing-like restarts to solve the vrptw. *European Journal of Operational Research* 150(1):115–127.

Mladenović, N., and Hansen, P. 1997. Variable neighborhood search. *Computers and Operations Research* 24(11):1097–1100.

Reinelt, G. 1991. Tsplib— traveling salesman problem library. *ORSA Journal on Computing* 3(4):376–384.

Taillard, E.; Badeau, P.; Gendreau, M.; Guertin, F.; and Potvin, J. Y. 1997. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science* 31(2):170–186.